Formalization of the Java_λ type system

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart Campus Horb
Department of Information Technology
Florianstraße 15, D–72160 Horb
m.pluemicke@hb.dhbw-stuttgart.de

Abstract. In Java 7 the language will be expanded by closures (λ–expressions) and function types.
In our contribution we give a formal definition for an abstract syntax of a reduced language Java_λ, define the type system, and formalize the subtyping relation.
We define the set of types as an extension of the generic type definition for Java 5 types.
Finally, we give type inference rules, which describe the typings of Java_λ expressions and statements and sketch a type inference algorithm.

1 Introduction

In several steps the Java type system is extended by features, which we know from functional programming languages. In Java 5.0 [GJSB05] generic types are introduced. Furthermore a reduced form of existential types (bounded wildcards) is introduced. For Java 7 it is announced, that function types should be introduced. Accordingly, closures (λ–expressions) should be introduced.
Type systems like this require to define syntactically large types. For example

```
Vector<? super Vector<? extends List<Integer>>>
```

or

$$\#\#\texttt{Matrix}(\#\texttt{Matrix}(\texttt{Matrix},\texttt{Matrix}))(\texttt{Matrix})$$

are correct types.
Considering all that, it is often rather inconvenient to give types like this, explictly. Furthermore it is often difficult for a programmer to decide whether such a complex type is the correct one for a given method or not.
This has caused us to develop a Java type inference system which assists the programmer by calculating types automatically. This type inference system allows us, to declare method parameters and local variables without type annotations. The type inference algorithm calculates the appropriate and principal types. In [Plü07] we presented a type inference algorithm for a core Java 5.0 language.
The extension of function types in Java are considered in several contributions. An early approach was given for PIZZA [OW97]. There are three newer approaches [BGGvdA,LLB,CS], which consider different aspects. Our approach is

following the Java language specification [lam10]. This approach is limited following Mark Reinhold's Blog (Principal Engineer Java SE and OpenJDK) to two key features:

- A literal syntax, for writing closures, and
- Function types, so that closures are first-class citizens in the type system.

Additionally, to integrate closures with the rest of the language and the platform two more features are needed:

- Closure conversion to implement a single-method interface or abstract class and
- Extension methods.

Our goal is to extend the type inference algorithm to the features of Java 7. For this we consider the type inference algorithm of Fuh and Mishra [FM88]. This algorithm infers types for a type system with subtyping and without additional overloading. The Java 7 preconditions are very similar.
The paper is organized as follows. In the next section we define an abstract syntax for a reduced language Java_λ. In the third section we consider the Java_λ types and the subtyping relation. In the fourth section we give the type inference rules. In the fifth section we consider the adaption of the Fuh and Mishra's type inference algorithm. Finally we close with a summary and an outlook.

2 The language

Source	:=	*class*∗
class	:=	Class(*stype*, [extends(*stype*),]*IVarDecl*∗, *FunDecl*∗)
IVarDecl	:=	InstVarDecl(*stype*, *var*)
FunDecl	:=	Fun(*fname*, [***type***], *lambdaexpr*)
block	:=	Block(*stmt*∗)
stmt	:=	*block* | Return(*expr*) | While(*bexpr*, *block*)
		| LocalVarDecl(*var*[, ***type***]) | If(*bexpr*, *block*[, *block*]) | *stmtexpr*
lambdaexpr	:=	Lambda(((*var*[, ***type***]))∗, (*stmt* | *expr*))
stmtexpr	:=	Assign(*var*, *expr*) | New(*stype*, *expr*∗) | Eval(*expr*, *expr*∗)
expr	:=	*lambdaexpr* | *stmtexpr* | this | This(*stype*) | super
		| LocalOrFieldVar(*var*) | InstVar(*expr*, *var*) | InstFun(*expr*, *fname*)
		| *bexp* | *sexp*

Fig. 1. The abstract syntax of Java_λ

The language Java_λ (Fig. 1) is an an abstract representation of a core of Java 7. It is an extension of our language in [Plü07]. The additional features are underlined. Beside instance variables functions can be declared in classes. A function

is declared by its name, its type, and the λ–expression. Methods are not considered in this framework. A λ–expression consists of an optionally typed variable and either an statement or an expression. Furthermore the statement expressions respectively the expressions are extended by evaluation-expressions, the λ–expressions, and instances of functions.
The concrete syntax in this paper is oriented at the syntax of [lam10].
The optional type annotations [***type***] are the types, which can be inferred by the type inference algorithm.

3 Types and subtyping

As a base for the type inference algorithm we have to make a formal definition of the Java 7 types. First we give again the definition of simple types (first-order types). The definition is connected to the corresponding definitions in [GJSB05], Section 4.5. and [Plü07], Section 2.

Definition 1 (Simple types). *Let $BTV^{(ty)}$ be the set of bounded type variables and TC a $(BTV)^*$–indexed set of type constructors (class names). Then, the set of* simple types $\mathsf{SType}_{TS}(BTV)$ *for the given type signature* $(\mathsf{SType}_{TS}(BTV), TC)$ *is defined as the smallest set satisfying the following conditions:*

- *For each type ty:* $\underline{BTV^{(ty)}} \subseteq \mathsf{SType}_{TS}(BTV)$
- $\underline{TC^{()}} \subseteq \mathsf{SType}_{TS}(BTV)$
- *For* $ty_i \in \mathsf{SType}_{TS}(BTV)$
  $\cup \{ \texttt{?} \}$
  $\cup \{ \texttt{? extends } \tau \mid \tau \in \mathsf{SType}_{TS}(BTV) \}$
  $\cup \{ \texttt{? super } \tau \mid \tau \in \mathsf{SType}_{TS}(BTV) \}$
  and $C \in TC^{(a_1|_{b_1} \ldots a_n|_{b_n})}$ *holds*

  $$\underline{C\texttt{<}ty_1, \ldots, ty_n\texttt{>}} \in \mathsf{SType}_{TS}(BTV)$$

  if after $C\texttt{<}ty_1, \ldots, ty_n\texttt{>}$ *subjected to the capture conversion resulting in the type* $C\texttt{<}\overline{ty_1}, \ldots, \overline{ty_n}\texttt{>}$[1]*, for each actual type argument* $\overline{ty_i}$ *holds:*

  $$\overline{ty_i} \leq^* b_i[a_j \mapsto \overline{ty_j} \mid 1 \leqslant j \leqslant n],$$

  where $\leq^*$ *is a subtyping ordering (Def. 3).*
- *The* *set of implicit type variables with lower or upper bounds* *belongs to* $\mathsf{SType}_{TS}(BTV)$

Simple types are the first-order base-types of the Java_λ type system. Simple types are used in Java_λ explictly in the extension relation, which defines the subtyping ordering.
The set of simple types is extended to the set of Java_λ types by adding function types.

[1] For non wildcard type arguments the capture conversion $\overline{ty_i}$ equals ty_i

Definition 2 (Types). *Let* $\mathsf{SType}_{TS}(\,BTV\,)$ *be a set of simpletypes. The set of* Java_λ *types* $\mathsf{Type}_{TS}(\,BTV\,)$ *is defined by*

- $\mathsf{SType}_{TS}(\,BTV\,) \subseteq \mathsf{Type}_{TS}(\,BTV\,)$
- *For* $ty, ty_i \in \mathsf{Type}_{TS}(\,BTV\,)$

$$\#\,ty\,(ty_1, \ldots, ty_n) \in \mathsf{Type}_{TS}(\,BTV\,)^2$$

Analogously, the definition of the subtyping relation on simple types is extended to the subtyping relation on Java_λ types.

Definition 3 (Subtyping relation $\leq^*$ on $\mathsf{SType}_{TS}(\,BTV\,)$).
Let $TS = (\mathsf{SType}_{TS}(\,BTV\,),\ TC)$ *be a type signature of a given* Java *program and* $<$ *the corresponding extends relation. The* subtyping relation $\leq^*$ *is given as the reflexive and transitive closure of the smallest relation satisfying the following conditions:*

- *if* $\theta < \theta'$ *then* $\theta \leq^* \theta'$.
- *if* $\theta_1 \leq^* \theta_2$ *then* $\sigma_1(\,\theta_1\,) \leq^* \sigma_2(\,\theta_2\,)$ *for all substitutions* $\sigma_1,\ \sigma_2 : BTV \to \mathsf{SType}_{TS}(\,BTV\,)$*, where for each type variable* a *of* θ_2 *holds* $\sigma_1(\,a\,) = \sigma_2(\,a\,)$ (*soundness condition*).
- $a \leq^* \theta_i$ *for* $a \in BTV^{(\theta_1 \& \ldots \& \theta_n)}$ *and* $1 \leqslant i \leqslant n$
- *It holds* $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \leq^* C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$ *if for each* θ_i *and* θ'_i*, respectively, one of the following conditions is valid:*
  - $\theta_i = {}_?\overline{\theta}_i$, $\theta'_i = {}_?\overline{\theta}'_i$ *and* $\overline{\theta}_i \leq^* \overline{\theta}'_i$.
  - $\theta_i = {}^?\overline{\theta}_i$, $\theta'_i = {}^?\overline{\theta}'_i$ *and* $\overline{\theta}'_i \leq^* \overline{\theta}_i$.
  - $\theta_i, \theta'_i \in \mathsf{SType}_{TS}(\,BTV\,)$ *and* $\theta_i = \theta'_i$
  - $\theta'_i = {}_?\theta_i$
  - $\theta'_i = {}^?\theta_i$

  (*cp. [GJSB05] §4.5.1.1 type argument containment*)
- *Let* $C\texttt{<}\overline{\theta}_1, \ldots, \overline{\theta}_n\texttt{>}$ *be the capture conversions of* $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>}$ *and* $C\texttt{<}\overline{\theta}_1, \ldots, \overline{\theta}_n\texttt{>} \leq^* C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$ *then holds* $C\texttt{<}\theta_1, \ldots, \theta_n\texttt{>} \leq^* C\texttt{<}\theta'_1, \ldots, \theta'_n\texttt{>}$.
- $\theta \leq^* {}_\theta|T$

Definition 4 (Subtyping relation $\leq^*$ on $\mathsf{Type}_{TS}(\,BTV\,)$). *Let* $\leq^*$ *be a subtyping relation on simple types* $\mathsf{SType}_{TS}(\,BTV\,)$*. Then, the continuation on* $\mathsf{Type}_{TS}(\,BTV\,)$ *is defined as:*

$$\#\,\theta\,(\theta'_1, \ldots, \theta'_n) \leq^* \#\,\theta'\,(\theta_1, \ldots, \theta_n) \quad \textit{iff} \quad \theta \leq^* \theta' \textit{ and } \theta_i \leq^* \theta'_i.$$

Example 1. Let the following Java_λ program be given.

```
class Matrix extends Vector<Vector<Integer>> {

  ##Matrix(#Matrix(Matrix, Matrix))(Matrix)
  op = #(Matrix m)(#(#Matrix(Matrix, Matrix) f)(f(Matrix.this, m)))
```

[2] Often function types $\#\,ty\,(ty_1, \ldots, ty_n)$ are written as $(ty_1, \ldots, ty_n) \to ty$.

```
  #Matrix(Matrix, Matrix)
  mul = #(Matrix m1, Matrix m2)
    (Matrix ret = new Matrix ();
      for(int i = 0; i < size(); i++) {
          Vector<? extends Integer> v1 = m1.elementAt(i);
          Vector<Integer> v2 = new Vector<Integer> ();
          for (int j = 0; j < v1.size(); j++) {
            int erg = 0;
            for (int k = 0; k < v1.size(); k++) {
                erg = erg + v1.elementAt(k)
                    * (m2.elementAt(k)).elementAt(j);
            }
            v2.addElement(erg);
          }
        ret.addElement(v2);
      }
    return ret;)

  public static void main(String[] args) {
    Matrix m1 = new Matrix(...);
    Matrix m2 = new Matrix(...);
    m1.op.(m2).(m1.mul);}
}
```

`op` is a curried function with two arguments. The first one is a matrix and the second one is a function which takes two matrices and results another matrix. The function `op` applies its second argument to its own object and its first argument.
`mul` is the ordinary matrix multiplication in closure representation.
Finally, in `main` the function `op` of the matrix `m1` is applied to the matrix `m2` and the function `mul` of `m1`.
From $\texttt{Matrix} \leq^* \texttt{Vector<Vector<Integer>>}$ follows

$$\begin{aligned} &\#\,\texttt{Matrix}\,(\texttt{Vector<Vector<Integer>>}, \texttt{Vector<Vector<Integer>>}) \\ &\quad \leq^* \#\,\texttt{Vector<Vector<Integer>>}\,(\texttt{Matrix}, \texttt{Matrix}). \end{aligned}$$

4 Type inference rules

In this paper we consider only type inference for function declarations *FunDecl*.
For the type inference system we need some additional definitions:
A set of *type assumptions* O is a map indexed by class names, which maps function names to types (e.g. $O_{\texttt{Matix}} = \{\, \texttt{mul} : \#\,\texttt{Matrix}\,(\texttt{Matrix}, \texttt{Matrix}) \,\}$).
In the following σ denotes a *substitution*, which substitutes some (bounded) type variables in a type.
Finally, we need two implications $\rhd_{Expr}$ and $\rhd_{Stmt}$. $(O, \tau, \tau') \rhd_{Expr} \; exp : \theta$ means that under the type assumptions O in the class τ, which direct superclass is τ', the expression *exp* has the type θ.

$(O, \tau, \tau') \rhd_{Stmt} \; stmt : \theta$ means that under the type assumptions O in the class τ, which direct superclass is τ' the statement *stmt* has the type θ.
First, we consider the type inference rules for λ–expressions. As the body of the λ–expressions either can be a statement or an expression, two rules are necessary.

$$[\mathbf{lambda}_{stmt}] \frac{(O \cup \{ x_i : \theta_i \}, \#\theta(\theta_1, \ldots, \theta_n), \texttt{Object}) \rhd_{Stmt} \; s : \theta}{(O, \tau, \tau') \rhd_{Expr} \; \mathsf{Lambda}(\,(x_1, \ldots, x_n), s\,) : \#\theta(\theta_1, \ldots, \theta_n)}$$

$$[\mathbf{lambda}_{expr}] \frac{(O \cup \{ x_i : \theta_i \}, \#\theta(\theta_1, \ldots, \theta_n), \texttt{Object}) \rhd_{Expr} \; e : \theta}{(O, \tau, \tau') \rhd_{Expr} \; \mathsf{Lambda}(\,(x_1, \ldots, x_n), e\,) : \#\theta(\theta_1, \ldots, \theta_n)}$$

Fig. 2. λ–expression rules

As λ–expressions are implemented as inner classes, the type of the class of the λ–expression is given as the type of the λ–expression itself.

For all statements of Java_λ there is a type inference rule. We give the most important rules in Fig. 3.

$$[\mathbf{Return}] \; \frac{(O, \tau, \tau') \rhd_{Expr} \; e : \theta}{(O, \tau, \tau') \rhd_{Stmt} \; \mathsf{Return}(\,e\,) : \theta}$$

$$[\mathbf{BlockInit}] \; \frac{(O, \tau, \tau') \rhd_{Stmt} \; stmt : \theta}{(O, \tau, \tau') \rhd_{Stmt} \; \mathsf{Block}(\,stmt\,) : \theta}$$

$$[\mathbf{Block}] \; \frac{\begin{array}{c}(O, \tau, \tau') \rhd_{Stmt} \; s_1 : \theta, \; (O, \tau, \tau') \rhd_{Stmt} \; \mathsf{Block}(\,s_2; \ldots; s_n;\,) : \theta' \\ \overline{\theta} \in \mathbf{MUB}(\theta, \theta')\end{array}}{(O, \tau, \tau') \rhd_{Stmt} \; \mathsf{Block}(\,s_1; s_2; \ldots; s_n;\,) : \overline{\theta}}$$

$$[\mathbf{If}] \; \frac{\begin{array}{c}(O, \tau, \tau') \rhd_{Stmt} \; \mathsf{Block}(\,B_1\,) : \theta, (O, \tau, \tau') \rhd_{Stmt} \; \mathsf{Block}(\,B_2\,) : \theta' \\ (O, \tau, \tau') \rhd_{Expr} \; e : \texttt{boolean}, \;\; \overline{\theta} \in \mathbf{MUB}(\theta_1, \theta_2)\end{array}}{(O, \tau, \tau') \rhd_{Stmt} \; \mathsf{If}(\,e, \mathsf{Block}(\,B_1\,), \mathsf{Block}(\,B_2\,)\,) : \overline{\theta}}$$

Fig. 3. Statement Rules

The type of a block of statements is basically defined by the type of the expression of the finishing Return-statement. The **MUB**-function determines the types of minimal upper bounds in the subtyping ordering.

The not presented statements Assign, New, and Eval have the type `void`, as no result is returned.

For all expressions of Java_λ there is a type inference rule. In addition to Fig. 2 we give the most important rules in Fig. 4. In different rules σ is a substitution, which substitutes (bounded) type variables by type-correct expressions.

$$[\textbf{Assign}]\ \frac{(O,\tau,\tau') \rhd_{Expr} e_1 : \theta', (O,\tau,\tau') \rhd_{Expr} e_2 : \theta}{(O,\tau,\tau') \rhd_{Expr} \mathsf{Assign}(e_1, e_2) : \theta'} \theta \leq^* \theta'$$

$$[\textbf{This}]\ \frac{}{(O,\tau,\tau') \rhd_{Expr} \mathsf{this} : \sigma(\tau)}$$

$$[\textbf{This\_enc}]\ \frac{}{(O,\tau,\tau') \rhd_{Expr} \mathsf{This}(\theta) : \sigma(\theta)}$$

$$[\textbf{Super}]\ \frac{}{(O,\tau,\tau') \rhd_{Expr} \mathsf{super} : \sigma(\tau')}$$

$$[\textbf{InstVar}]\ \frac{(O,\tau,\tau') \rhd_{Expr} re : \sigma(\overline{\theta}), \quad (v : \theta) \in O_{\overline{\theta}}}{(O,\tau,\tau') \rhd_{Expr} \mathsf{InstVar}(re, v) : (\sigma' \circ \sigma)(\theta)}$$

$$[\textbf{InstFun}]\ \frac{(O,\tau,\tau') \rhd_{Expr} re : \sigma(\overline{\theta}), \quad (f : \#\,\theta\,(\theta_1, \ldots, \theta_n)) \in O_{\overline{\theta}}}{(O,\tau,\tau') \rhd_{Expr} \mathsf{InstFun}(re, f) : (\sigma' \circ \sigma)(\#\,\theta\,(\theta_1, \ldots, \theta_n))}$$

Fig. 4. Expression rules

The **Assign**-rule is canonically defined.
The **This**–rule types the expression this by its class τ. As each λ–expression is implemented as an anonymous inner class, τ is the type of the most nested λ–expression. In contrast the **This_enc**–rule types the expression this of an enclosing class θ.
The rules InstVar and InstFun types identifiers which are defined in a class $\overline{\theta}$ as fields and functions, respectively.

5 Type inference algorithm

In the late eighties Fuh and Mishra [FM88] gave an algorithm for type inference in the λ-calculus with subtyping. Their calculus indeed allows subtyping but no additional overloading. This means that the type system of Java_λ and the

type system of their language are equivalent. There is only one small difference: Their subtyping relations are finite, while in Java_λ infinite chains are possible in subtyping relations. We considered this problem in [Plü09]. The result is that there is finite number of representants for the infinite chains. We will use this result in the type inference algorithm.

5.1 Summary of Fuh and Mishra's algorithm

The algorithm is called WTYPE, which stands for *well-typing*. A *well-typing* is data-structure $C, A \vdash N : t$, where C is a set of consistence coercions (unsolved unequations), A is a set of type assumptions, N is an expression, and t is the derived type. The algorithm has the following signature:

$$\mathbf{WTYPE} : \texttt{TypeAssumptions} \times \texttt{Expression} \to \texttt{WellTyping} + \{\, \mathit{fail} \,\}$$

This means the result of the algorithm is a type with a set of constraints. **WTYPE** consists of four functions.

TYPE : $\texttt{TypeAssumptions} \times \texttt{Expression} \to \texttt{Type} \times \texttt{CoercionSet}$ maps a fresh type variable to each subterm of the input expression and determines coercions, which contain the function and the tuple constructors derived from the structure of the λ–expression.

MATCH : $\texttt{CoercionSet} \to \texttt{Substitution} + \{\, \mathit{fail} \,\}$ determines a substitution σ. σ applied to the input coercion set C results in a minimal matching instance of C. The algorithm **Match** is an adoption to coercions of the Martelli, Montanari unification algorithm [MM82], which is used in the Damas Milner type inference algorithm [DM82].

SIMPLIFY : $\texttt{CoercionSet} \to \texttt{AtomicCoercionSet}$ eliminates the type constructors, especially the *function* and the *tuple-constructor*.

CONSISTENT : $\texttt{AtomicCoercionSet} \to \texttt{Boolean} + \{\, \mathit{fail} \,\}$ checks if a set of atomic coercions is consistent, by determining all possible instances of each variable. If finally for each variable there is a non-empty set of instances, the set of atomic coercions is consistent.

This leads to the following algorithm:

$$\begin{aligned}
\mathbf{WTYPE}(\, A, e \,) = {} & \mathbf{let}\ (\theta, C) = \mathbf{TYPE}(\, A, e \,)\ \mathbf{in} \\
& \mathbf{let}\ \sigma = \mathbf{MATCH}(\, C \,)\ \mathbf{in} \\
& \mathbf{let}\ C' = \mathbf{SIMPLIFY}(\, \sigma(\, C \,) \,)\ \mathbf{in} \\
& \mathbf{if}\ \mathbf{CONSISTENT}(\, C' \,)\ \mathbf{then} \\
& \quad (C', A) \vdash e : \sigma(\, \theta \,) \\
& \mathbf{else} \\
& \quad \mathit{fail}
\end{aligned}$$

5.2 Adaption to the Java_λ type system

Now we give a sketch how to adapt the type inference algorithm to Java_λ λ–expressions.
The functions **SIMPLIFY** and **CONSISTENT** have to be changed.

SIMPLIFY: The functions of the unification algorithm of Martelli and Montanari [MM82] are substituted by the corresponding functions of the Java type unification algorithm [Plü09].

CONSISTENCE: In the functions $\uparrow$ and $\downarrow$, which determine iteratively all supertypes and all subtypes, respectively, of a set of types, the functions `greater` and `smaller` from [Plü09] are used, such that only a finite set of representants is determined.

We give an example, how to use the adopted algorithm.
Let the program from 1 be given again, where the function `op` is not explictly typed:

```
op = #(m)(#(f)(f(Matrix.this, m)))
```

TYPE($\emptyset$, `#(m)(#(f)(f(Matrix.this, m)))`) $= (t_{\mathtt{op}}, C)$, where
$C = \{\, (t_m \to t_{\#f} \lessdot t_{\mathtt{op}}), (t_f \to t_{f(\,M.this,m\,)} \lessdot t_{\#f}), (t_f \lessdot (t_1, t_2) \to t_3),$
$(\mathtt{Matrix} \lessdot t_1), (t_m \lessdot t_2), (t_3 \lessdot t_{f(\,M.this,m\,)}\,\}$)

MATCH(C):

- From $(t_m \to t_{\#f} \lessdot t_{\mathtt{op}})$ follows

$$\sigma_1 = \{\, (t_{\mathtt{op}} \mapsto \beta \to \beta')\,\}.$$

- From $(t_f \to t_{f(\,M.this,m\,)} \lessdot t_{\#f})$ follows

$$\sigma_2 = \sigma_1 \cup \{\, (t_{\#f} \mapsto \gamma_1 \to \gamma_1')\,\}.$$

- From $(t_f \lessdot (t_1, t_2) \to t_3)$ follows

$$\sigma_3 = \sigma_2 \cup \{\, (t_f \mapsto (\epsilon_1, \epsilon_1') \to \epsilon_1'')\,\}.$$

- From $\sigma_3(\, t_f \to t_{f(\,M.this,m\,)} \lessdot t_{\#f}\,) =$
$(((\epsilon_1, \epsilon_1') \to \epsilon_1'') \to t_{f(\,M.this,m\,)} \lessdot \gamma_1 \to \gamma_1')$ follows

$$\sigma_4 = \sigma_3 \cup \{\, (\gamma_1 \mapsto (\epsilon_2, \epsilon_2') \to \epsilon_2'')\,\}.$$

- From $\sigma_4(\, t_m \to t_{\#f} \lessdot t_{\mathtt{op}}\,) = \; t_m \to ((\epsilon_2, \epsilon_2') \to \epsilon_2'') \to \gamma_1' \lessdot \beta \to \beta'$ follows

$$\sigma_5 = \sigma_4 \cup \{\, \beta' \mapsto ((\epsilon_3, \epsilon_3') \to \epsilon_3''') \to \gamma_2'\,\}.$$

Result: $\sigma = \{\, (t_{\mathtt{op}} \mapsto \beta \to ((\epsilon_3, \epsilon_3') \to \epsilon_3'') \to \gamma_2'),$
$(t_{\#f} \mapsto ((\epsilon_2, \epsilon_2') \to \epsilon_2'') \to \gamma_1'), (t_f \mapsto (\epsilon_1, \epsilon_1') \to \epsilon_1'')\,\}$

It holds: $\sigma(\, C\,) = \{\quad t_m \to ((\epsilon_2, \epsilon_2') \to \epsilon_2'') \to \gamma_1' \lessdot \beta \to ((\epsilon_3, \epsilon_3') \to \epsilon_3'') \to \gamma_2',$
$((\epsilon_1, \epsilon_1') \to \epsilon_1'') \to t_{f(\,this,m\,)} \lessdot ((\epsilon_2, \epsilon_2') \to \epsilon_2'') \to \gamma_1',$
$(\epsilon_1, \epsilon_1') \to \epsilon_1'' \lessdot (t_1, t_2) \to t_3,$
$\mathtt{Matrix} \lessdot t_1,$
$t_m \lessdot t_2,$
$t_3 \lessdot t_{f(\,this,m\,)}\,\}$

SIMPLIFY$(\,\sigma(\,C\,)\,) = C'$, where

$$\begin{aligned} C' = \{\, & \beta \lessdot t_m, \epsilon_2 \lessdot \epsilon_3, \epsilon'_2 \lessdot \epsilon'_3, \epsilon''_3 \lessdot \epsilon''_2, \gamma'_1 \lessdot \gamma'_2 \\ & \epsilon_1 \lessdot \epsilon_2, \epsilon'_1 \lessdot \epsilon'_2, \epsilon''_2 \lessdot \epsilon''_1, t_{f(\,this,m\,)} \lessdot \gamma'_1 \\ & t_1 \lessdot \epsilon_1, t_2 \lessdot \epsilon'_1, \epsilon''_1 \lessdot t_3 \\ & \texttt{Matrix} \lessdot t_1 \\ & t_m \lessdot t_2 \\ & t_3 \lessdot t_{f(\,this,m\,)} \,\} \end{aligned}$$

CONSISTENT$(\,C'\,)$**:** For each $\theta \lessdot \theta' \in C'$ we have to determine I_θ respectively $I_{\theta'}$. If all $I_\theta \neq \emptyset$ the atomic coercion set is consistent.

In the following table we consider the iteration steps[3].

It	$Coercion$	$I_{\texttt{Matrix}}$	I_{t_1}	I_{ϵ_1}	I_{ϵ_2}	I_{ϵ_3}	...
0		M	$*$	$*$	$*$	$*$	$*$
1	M $\lessdot t_1$	M	M, V<V<Int>>	$*$	$*$	$*$	$*$
1	$t_1 \lessdot \epsilon_1$	M	M, V<V<Int>>	M, V<V<Int>>	$*$	$*$	$*$
1	$\epsilon_1 \lessdot \epsilon_2$	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	$*$	$*$
1	$\epsilon_2 \lessdot \epsilon_3$	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	$*$
1	...	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	$*$
2	...	M	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	M, V<V<Int>>	$*$

This means **CONSISTENT**$(\,\sigma(\,C'\,)\,) =$ `true`.

From this follows **WTYPE**$(\,\emptyset,$ `#(m)(#(f)(f(Matrix.this, m)))` $) =$

$$(C', \emptyset) \vdash \texttt{\#(m)(\#(f)(f(Matrix.this, m)))} : \beta \to ((\epsilon_3, \epsilon'_3) \to \epsilon''_3) \to \gamma'_2.$$

As in the function CONSISTENCE all possible instances are determined, it holds:

$$\epsilon_3 = \texttt{Matrix} \text{ or } \texttt{Vector<Vector<Integer>>}.$$

Furthermore from C' follows, that it holds:

- $\beta \lessdot t_m \lessdot t_2 \lessdot \epsilon'_1 \lessdot \epsilon'_2 \lessdot \epsilon'_3$
- $\epsilon''_3 \lessdot \epsilon''_2 \lessdot \epsilon''_1 \lessdot t_3 \lessdot t_{f(\,this,m\,)} \lessdot \gamma'_1 \lessdot \gamma'_2$,

which describe corelations of type variables of the result type.

6 Conclusion and future work

We have considered the Java 7 extensions closures and function types as first-class citizens. We gave an abstract definition of the subtyping relation and define the type inference rules for a small core language Java_λ. The properties of the Java_λ type system are very similar to the type system, which is considered by

[3] We consider only the non-wildcard types.

Fuh and Mishra [FM88]. Therefore, we gave a sketch of the adoption of their type inference algorithm to Java_λ.
The result of Fuh and Mishra's algorithm are well-typings $C, A \vdash N : t$, where C is a set of consistence coercions (unsolved unequations), A is a set of type assumptions, N is an expression, and t is the derived type. For an implementation of the type inference algorithm in a Java IDE the realization of well-typings is an open problem, as the Java type systems indeed allows bounded type variables, but no constraints on type variables.
Furthermore we have to prove correctness and soundness of the Java_λ adopted type inference algorithm.
Finally, after solving the problem of well-typing realization, we have to implement the algorithm.

References

[BGGvdA] Gilad Bracha, Neal Gafter, James Gosling, and Peter von der Ahé. Closures for the java programming language (aka BGGA). `http://www.javac.info`.

[CS] Stephen Colebourne and Stefan Schulz. First-class methods: Java-style closures (aka FCM). `http://docs.google.com/Doc?id=ddhp95vd_6hg3qhc`.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. *Proc. 9th Symposium on Principles of Programming Languages*, 1982.

[FM88] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Proceedings 2nd European Symposium on Programming (ESOP '88)*, pages 94–114, 1988.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification*. The Java series. Addison-Wesley, 3rd edition, 2005.

[lam10] Project lambda: Java language specification draft. `http://mail.openjdk.java.net/pipermail/lambda-dev/attachments/20100212/af8d2cc5/attachment-0001.txt`, 2010. Version 0.1.5.

[LLB] Bob Lee, Doug Lea, and Josh Bloch. Concise instance creation expressions: Closures without complexity (aka CICE). `http://docs.google.com/Doc.aspx?id=k73_1ggr36h`.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[Plü07] Martin Plümicke. Typeless Programming in Java 5.0 with wildcards. In Vasco Amaral, Luís Veiga, Luís Marcelino, and H. Conrad Cunningham, editors, *5th International Conference on Principles and Practices of Programming in Java*, volume 272 of *ACM International Conference Proceeding Series*, pages 73–82, September 2007.

[Plü09] Martin Plümicke. Java type unification with wildcards. In Dietmar Seipel, Michael Hanus, and Armin Wolf, editors, *17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Wrzburg, Germany, October 4-6, 2007, Revised Selected Papers*, volume 5437 of *Lecture Notes in Artificial Intelligence*, pages 223–240. Springer-Verlag Heidelberg, 2009.